DMLAP Final Project – Report

Sound space exploration using corpus-based concatenative synthesis and assisted interactive machine learning

Max Graf – mgraf001

Introduction / background

This project deals with the problem of *sound space exploration*. Sound spaces can be defined as a collection of points describing arbitrary sonic attributes. They can be constructed using timbral descriptors of music, audio features of sound files, synthesizer parameters, and many more.
The two main use cases for sound spaces are inspection (visualising a collection of sounds) and synthesis (creating new audio from an existing corpus of sounds). This work focuses on the second aspect. Before I explain exactly *how* the synthesis works, I will elucidate *what* is actually synthesised.

Corpus-based concatenative synthesis

I employ *corpus-based concatenative synthesis* (CBCS), a grain-based technique described in [2]. Here, a (large) collection of source audio data, e.g. a sample library, is split up into short chunks of 1-20ms of length (***grains***). Each grain is subjected to sonic analysis. Depending on the use case of the system, the method of analysis can vary greatly. My analysis consists of the extraction of *audio features*, such as loudness, pitch, spectral centroid[1] and spectral flux[2]. A grain is thus defined through

1. *Source*: A reference to the source audio file
2. *Start index*: An index pointing to the exact position in the source audio file where the grain *starts*
3. *Grain length*: The length of the grain, expressed in number of audio samples
4. *Audio features*: The audio features described above

This information is stored in a database and then recombined in a systematic manner in order to create new sonic material. This raises the question of exactly *how* this recombination takes place.

Assisted interactive machine learning

In this project I used a technique called *assisted interactive machine learning* (AIML) to aid in the recombination of grains from the database. AIML was first described by Scurto et al. in [4] and further developed by Visi and Tanaka in [3]. In AIML, a deep reinforcement learning agent is iteratively trained to learn a set of parameters. The learning process happens *in conjunction with a user*, who regularly provides feedback to the agent. The agent uses this feedback to further refine its decision on *how the parameters should be tweaked further in the future*.
In [3] and [4], the parameters the agent learns are linked to a software synthesizer. In my work, they are linked to *audio features* of grains, more specifically to a *sequence* of audio features of grains. This means that the agent learns a ***trajectory*** through the grain space. User feedback is *binary* (like/dislike). Positive feedback solidifies the agent's choice of parameters, negative feedback encourages the agent to change its parameters.

[1] https://en.wikipedia.org/wiki/Spectral_centroid, accessed on 10 April 2021
[2] https://en.wikipedia.org/wiki/Spectral_flux, accessed on 10 April 2021

Implementation

The project consists of three distinct pieces of software. The *backend* component is a C++/JUCE[3] application. It is responsible for creating and managing the database of grains, recording and analysing audio, and converting the parameters of the agent to a concrete sequence of grains, and thereby, audio. The *essentia*[4] C++ library was used for audio feature extraction. The SQLite[5] database system was used for the database.

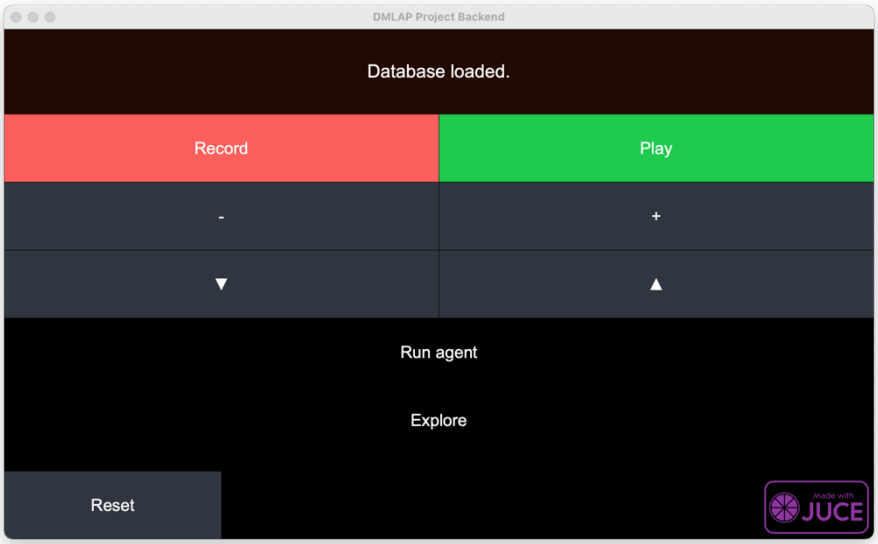

Figure 1: User interface of the JUCE application

The backend offers a user interface that can be used to control the agent, shown in figure 1. It communicates with the agent via Open Sound Control (OSC) messages.
The second component is the *agent framework*, implemented in Python and TensorFlow. It contains routines for running, updating and resetting the agent.
The third component is a *mobile web interface* that can be used optionally to interact with the agent. It provides additional functionality for interacting with generated trajectories. The web interface is based on the Node.js framework (server) and p5.js (frontend). The socket.io library was used to send data from the client (running in a web browser on a mobile phone) to the Node.js server.

Creative motivation & intended use

I have been intrigued by grain-based synthesis methods for a long time. Constructing (partly or completely) new sounds from an existing corpus of data (such as seen in [1]) is a fascinating concept for me, and when I read Visi and Tanaka's paper [3] a few months ago I wanted to see how well it applied to CBCS. Currently I see two main use cases for this work:

1. Music production: The code natively compiles to an audio plugin format, which can then be used in Ableton Live, Logic, etc. This would allow (electronic) musicians to re-imagine their songs, sample libraries, etc. and embed them seamlessly in their musical environment. For example, I really enjoy the idea of someone using the same sounds for a drum part, and then, through this system, as an arpeggiated, synthesizer-like element.
2. Performance: The mobile web interface decouples the user from the machine running the backend. It exposes the full functionality of the backend, making it possible to play and interact with the agent on a stage, for example.

Limitations

Currently, there are several limitations to the system:

[3] https://juce.com/, accessed on 11 April 2021
[4] https://essentia.upf.edu/, accessed on 11 April 2021
[5] https://www.sqlite.org/index.html, accessed on 11 April 2021

1. Source material size: I tested the system with a maximum of 10GB of source material (**.wav* files). This is due to the in-memory management of the database, as well as the performance of the SQLite database.
2. Audio features: The audio features I selected are based on my knowledge of grain-based synthesis techniques. This does not mean that they are the most well-suited for this task. Based on the desired aesthetic of the generated sound, different audio features could be used for grain matching.
3. Agent feedback. Due to the binary nature of the agent feedback, it is impossible to guarantee that the agent will completely settle on certain parameters. This can lead to some frustration, as the agent will sometimes make a minute change to one of the parameters, resulting in the selection of an audibly different grain.
4. The system in its current form is limited to the macOS platform.

Conclusion

I am very satisfied with the outcome of this project. Through re-adjusting and tweaking the backend, as well as the agent, I reached a point at which interesting sounds can be found and solidified relatively quickly (20s – 1 minute). I also received increasingly positive feedback from another person (my flatmate, who played with the system as its development progressed), culminating in a prompt to send him the system as an audio plugin, so that he could use it in his own music production.
Future work could make the system more flexible, for example by pre-calculating several more audio features and letting users choose the audio features used for grain matching at runtime. The search for closest matching grains could be optimised by using specialised data structures, such as k-D trees for example. However, this would not solve the imposed limit on the maximum number of grains, since the data would still need to reside in the computer memory.

Third party resources

As stated above, I have used the JUCE framework and the essentia C++ library for audio feature extraction and the SQLite database system for database management. The remaining C++ code (all of the backend) is fully authored by me.
The code for the python agent is taken from the implementation published by Scurto et al.[6]. I changed the configuration of the RL agent to suit the problem at hand. This includes parameters such as the number of states of the agent, the depth of the hidden layers in the deep neural network as well as the agent's learning rate and replay size. In addition to that I adapted the reinforcement learning loop to suit the large number of parameters. My agent deals with a total of 132 parameters (floating point numbers), resulting from 4 audio features per grain and a trajectory length of 33 grains. Whereas in the original implementation one cycle changes one parameter at a time, I have modified it to iterate over all 132 parameters once to complete a cycle, which speeds up the process of finding a desired trajectory.
The code for the web interface is fully authored by me.

Compiling and running

Note: All instructions are for the macOS operating system.

Backend

I will include a macOS executable for the backend application in addition to the code. The CMake toolchain as well as the JUCE framework (version 6 or higher) are required to compile the code. With both installed the steps to compile the backend component are:

1. Open the *CMakeLists.txt* file located in the *DMLAP_Backend* folder. On line 26, change the path to point to your JUCE installation folder.
2. Open a terminal and navigate to the *DMLAP_Backend* folder.
3. Execute *cmake . -B cmake-build-dir* – This will create the CMake build tree.
4. Execute *cmake --build cmake-build-dir* – This will compile the code into an executable.

[6] https://github.com/Ircam-RnD/coexplorer, accessed on 11 April 2021

5. Navigate to the "*cmake-build-dir/DMLAP_Backend_artefacts*" folder and execute the "DMLAP Project Backend.app" file.
6. **NOTE:** If the backend application does not ask for microphone permissions on start-up (due to some strange macOS behaviour), you may have to start it via the terminal. This can be done by executing "*/path/to/the/backend/DMLAP\ Project\ Backend.app/Contents/MacOS/DMLAP\ Project\ Backend*"

Shortcuts:

In addition to buttons of the GUI in the backend, the keyboard can also be used to interact with the system more quickly:

- Spacebar: Triggers playback of generated trajectory
- A: Negative singular agent feedback
- D: Positive singular agent feedback
- S: Negative zone feedback – zone feedback encourages the agent to make a bigger change to its parameters.
- W: Positive zone feedback
- E: **Explore**: Reset the agent to a new trajectory through the grain space
- R: Record ~1s of audio from the computer microphone and *prime* the agent with the audio. This is useful at the start of the exploration to tell the agent roughly the kind of sound you want to achieve.
- P: Playback of recorded audio
- L: **Loop** the generated trajectory
- "." (dot key): **Run the agent**
- H: Show/hide audio device overlay: Use this to select your audio device, e.g. when using headphones
- Backspace: Resets everything, basically restarts the whole system (both backend and the agent) without having to close and re-open all applications.

Reinforcement learning agent

The agent requires ***Python version 3.6*** or above. The safest way to run the agent is to create a new virtual environment[7]. *Once the new virtual environment has been activated*, the agent is run via the following steps:

1. Open a terminal and navigate to the *coexplorer_dmlap* folder
2. Execute *pip install -r requirements.txt* – This will install all required python packages
3. Execute *python TheInteractiveAgent_V5.py* – This will start the agent framework. The deep reinforcement learning cycle will begin once the "Run agent" button is clicked in the UI of the JUCE backend application (both applications must run in parallel).

Mobile web interface

The mobile web interface requires a Node.js installation. The steps to execute it are:

1. Navigate to the *dmlap_project_frontend_mobile* folder.
2. In *app.js* on line 13, change the given IP address to your local IP address in your network. On macOS this can be obtained by going to System Preferences -> Network.
3. Open a terminal, navigate to the *dmlap_project_frontend_mobile* folder.
4. Execute *npm i* – This will install the required JavaScript packages.
5. Execute *node app.js* – This will start the Node.js server
6. On your phone (connected to the same local network), open the website *https://your_ip_address* where *your_ip_address* corresponds to the address you inserted in step 2.
7. Grant accelerometer permissions on your phone.

Note: In order for the mobile frontend to work, both the JUCE backend and the python agent must be running already.

[7] https://docs.python.org/3/tutorial/venv.html, accessed on 11 April 2021

Caveat: In case the "cert.pem" and "key.pem" files included in the submission do not work, i.e. your phone cannot obtain a secure *https* connection to the server, you must generate a new key and certificate. This can be done quickly using OpenSSL[8] and the command

openssl req -newkey rsa:2048 -new -nodes -keyout key.pem -out cert.pem

After the files have been generated replace the existing key.pem and cert.pem files in the *dmlap_project_frontend_mobile* folder and restart the Node.js server.

Bibliography

[1] J. Françoise, N. Schnell, R. Borghesi, and F. Bevilacqua, 'Probabilistic Models for Designing Motion and Sound Relationships', *Proceedings of the 2014 international conference on new interfaces for musical expression*, p. 6, 2014.

[2] M. Zbyszynski, B. Di Donato, F. Visi, and A. Tanaka, 'Gesture-Timbre Space: Multidimensional Feature Mapping Using Machine Learning & Concatenative Synthesis', in *Perception, Representations, Image, Sound, Music*, R. Kronland-Martinet, S. Ystad, and M. Aramaki, Eds. Springer International Publishing, 2021.

[3] F. G. Visi and A. Tanaka, 'Towards Assisted Interactive Machine Learning: Exploring Gesture-Sound Mappings Using Reinforcement Learning', presented at the International Conference on Live Interfaces (ICLI), Trondheim, Norway, Jul. 02, 2020, doi: 10.5281/zenodo.3928167.

[4] H. Scurto, B. V. Kerrebroeck, B. Caramiaux, and F. Bevilacqua, 'Designing Deep Reinforcement Learning for Human Parameter Exploration', *ACM Trans. Comput.-Hum. Interact.*, vol. 28, no. 1, p. 1:1-1:35, Jan. 2021, doi: 10.1145/3414472.

[8] https://www.openssl.org/, accessed on 11 April 2021